Communication efficient multi-processor FFT

Communication Efficient Multi-processor FFT

S. Lennart Johnsson
Michel Jacquemin
Robert L. Krawitz

TR-25-91

October 1991

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Communication Efficient Multi-processor FFT

S. Lennart Johnsson[1], Michel Jacquemin[2] and Robert L Krawitz

Thinking Machines Corp.
245 First Street,
Cambridge, MA 02142

Abstract

Computing the Fast Fourier Transform on a distributed memory architecture by a direct pipelined radix-2 algorithm, a bi-section or multi-section algorithm, all yield the same communications requirement, if communication for all FFT stages can be performed concurrently, the input data is in normal order, and the data allocation consecutive. With a cyclic data allocation, or bit-reversed input data and a consecutive allocation, multi-sectioning offers a reduced communications requirement by approximately a factor of two. For a consecutive data allocation, normal input order, a decimation-in-time FFT requires that $\frac{P}{N} + d - 2$ twiddle factors be stored for P elements distributed evenly over N processors, and the axis subject to transformation distributed over 2^d processors. No communication of twiddle factors is required. The same storage requirements hold for a decimation-in-frequency FFT, bit-reversed input order, and consecutive data allocation. The opposite combination of FFT type and data ordering requires a factor of $\log_2 N$ more storage for N processors.

The peak performance for a Connection Machine system CM-200 implementation is 12.9 Gflops/s in 32-bit precision, and 10.7 Gflops/s in 64-bit precision for unordered transforms local to each processor. The corresponding execution rates for ordered transforms are 11.1 Gflops/s and 8.5 Gflops/s, respectively. For distributed one- and two-dimensional transforms the peak performance for unordered transforms exceeds 5 Gflops/s in 32-bit precision, and 3 Gflops/s in 64-bit precision. Three-dimensional transforms executes at a slightly lower rate. Distributed ordered transforms executes at a rate of about $\frac{1}{2}$ to $\frac{2}{3}$ of the unordered transforms.

1 Introduction

The main contributions of this paper are communication efficient multi-processor algorithms for the Cooley-Tukey Fast Fourier Transform [2] (FFT). The impact on performance of different data layouts is evaluated and an implementation on the Connection Machine system CM-200 is described. The algorithms are efficient in their use of the communication system, in particular systems with processors interconnected as Boolean cube networks allowing concurrent communication on all channels of every processor. The algorithms are also efficient in the use of storage for twiddle factors with no communication of twiddles required, when the factors are precomputed. In a distributed memory architecture a poor choice of FFT

[1] Also affiliated with the Division of Applied Sciences, Harvard University.
[2] Present address Department of Computer Science, Yale University.

algorithm may require twiddle factors to be communicated, or the storage requirements may exceed the data storage requirement by a factor of $\log N$ for N processors. Finally, the algorithms are also efficient with respect to their use of the bandwidth between each processor and its memory.

The distribution of data among the memory modules in a distributed memory architecture has a significant impact on performance. We briefly discuss this issue for both one-dimensional and multi-dimensional transforms.

It is well known that the Cooley-Tukey, *in-place* FFT reorders the data, such that after the transform the component in location $i = (i_{p-1}i_{p-2}\ldots i_0)$ has index $(i_0 i_1 \ldots i_{p-2}i_{p-1})$. The output index is the *bit-reversed* value of the input index. An FFT that leaves the output data in this order is *unordered*. An *ordered* FFT has the same data order for input and output.

The implementations being discussed fully utilize the communication system for the computations of the unordered FFT. All channels of every processor are used concurrently. The reordering required for an ordered transform is made explicitly. Reordering algorithms, and implementations thereof are discussed elsewhere [10, 11, 3]. No gain in communication efficiency is possible by interleaving the reordering with the FFT computations, when all channels are used for the unordered transform, unlike the case with communication restricted to one channel at-a-time [20, 22]. For reference, we include performance measurements both for unordered and ordered transforms.

The feasibility of different implementations of the Cooley-Tukey algorithm depends critically upon architectural characteristics. In the Connection Machine systems CM-2 and CM-200 the memory is distributed among up to 2048 floating-point processors. The maximum memory per processor is 4 Mbytes. In model CM-200, the floating-point processors support both 32-bit and 64-bit arithmetic. Data paths internal to the floating-point processors are 64-bits wide. Each processor has a single 32-bit wide data path to its local memory. The processors are interconnected as an 11-dimensional Boolean cube, with two communications channels between each pair of processors. Communication can be performed on all channels of every processor concurrently. The primitive communications operation is an exchange.

The Discrete Fourier Transform is defined by

$$X(l) = \sum_{j=0}^{P-1} \omega_P^{lj} x(j), \quad \forall l \in [0, P-1], \quad \omega_P = e^{-\frac{2\pi i}{P}}$$

and the Inverse Discrete Fourier Transform is defined by

$$X(l) = \frac{1}{N} \sum_{j=0}^{P-1} \omega_P^{-lj} x(j), \quad \forall l \in [0, P-1], \quad \omega_P = e^{-\frac{2\pi i}{P}}.$$

The Cooley-Tukey Fast Fourier Transform [2] evaluates these matrix vector products in $\log_2 P$ stages by recursively using a splitting formula of the type

$$
\begin{aligned}
X(l) &= \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l}\, x(2j') + \omega_P^l \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l}\, x(2j'+1) \\
X(l+\frac{P}{2}) &= \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l}\, x(2j') - \omega_P^l \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{j'l}\, x(2j'+1)
\end{aligned}
$$

for a forward *decimation-in-time* (DIT) FFT, or of the type

$$
\begin{aligned}
X(2l') &= \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{jl'}(x(j) + x(j+\frac{P}{2})) \\
X(2l'+1) &= \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{jl'}(\omega_P^j(x(j) - x(j+\frac{P}{2}))
\end{aligned}
$$

for a forward *decimation-in-frequency* (DIF) FFT. The coefficients ω_P^{lj} are known as *twiddle factors*. Both these types of FFT are known as radix-2 FFTs. The inverse transform can be computed in the same manner as the forward transform by using conjugate twiddle factors. Figure 1 shows a radix-2, DIT FFT, and Figure 2 shows a radix-2 DIF FFT. The difference of significance with respect to computing an FFT on a distributed data structure is that the DIF and DIT FFT use the twiddle factors in opposite orders. The DIT FFT use all twiddle factors in the last stage, while the DIF FFT use all twiddle factors in the first stage. Also, the twiddle factors for the DIF FFT are ordered in the same way as the input data, i.e., in normal order for normal order input data, while the twiddle factors for a DIT FFT are in bit-reversed order for normal order input. The consequences of these differences for FFT computations on data sets distributed throughout the memories of a multi-processor are discussed in Section 4.

For $P = R^m$ the splitting formulas can be generalized to a radix-R FFT. Figure 3 shows computational kernels corresponding to radix-4 DIT and DIF splitting formulas. Figure 4 shows the computational kernels corresponding to radix-8 DIT and DIF FFT. For details of the derivations see for instance [16, 17, 18].

As the radix of the FFT increases the number of arithmetic operations decreases somewhat. However, the main advantage from an increased radix in architectures with a limited memory bandwidth is a reduced need for memory accesses [5, 6]. The number of real operations (leading terms only) and memory accesses for radix-2, 4, and 8 kernels are given in Table 1. The number of arithmetic operations for the radix-8 algorithm is approximately 20% less than that of the radix-2 algorithm. The exact number of multiplications and additions can be found for instance in [16]. Whereas the reduction in arithmetic operations is modest a radix-8 FFT offers a reduction in the number of memory operations by a factor of almost three compared to a radix-2 algorithm. These kernel sizes are relevant for exploiting the

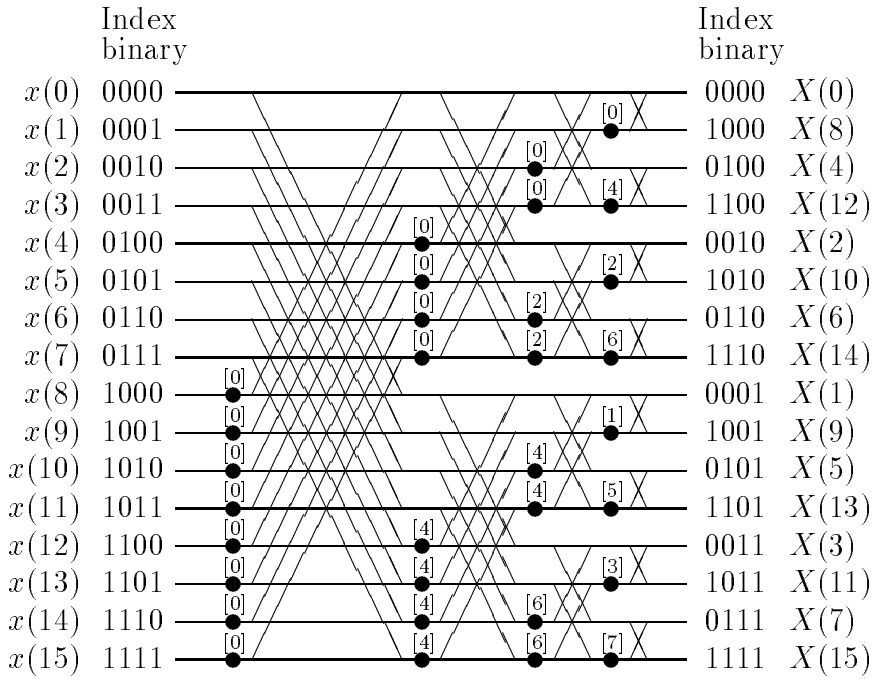

Figure 1: Decimation-in-time FFT.

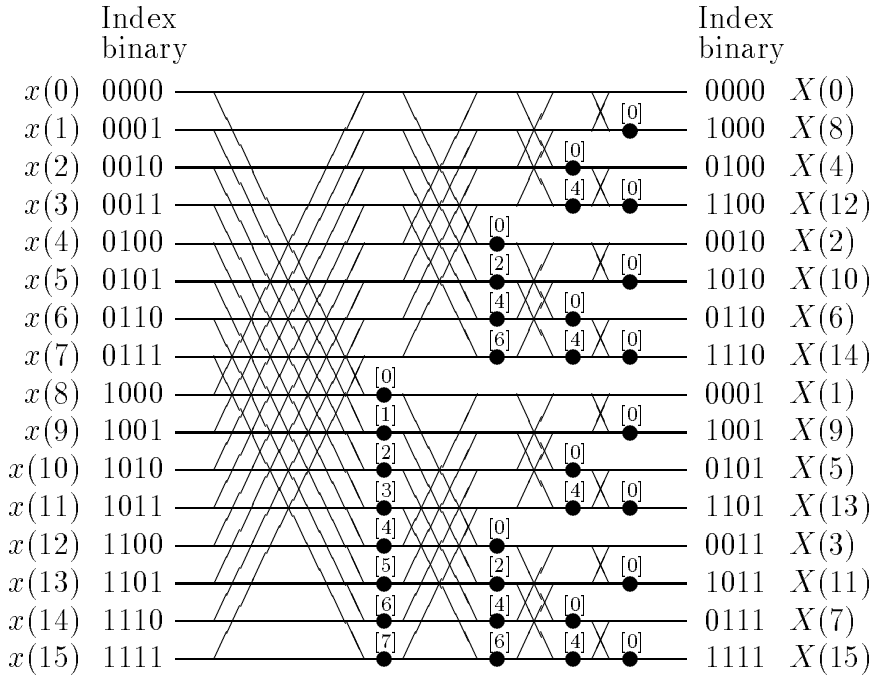

Figure 2: Decimation-in-frequency FFT.

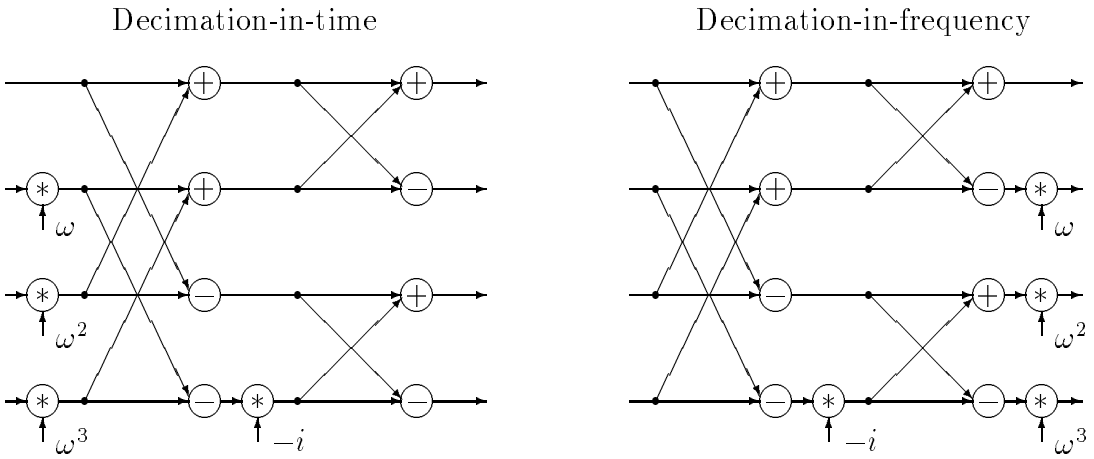

Figure 3: Radix-4 decimation-in-time and decimation-in-frequency kernels.

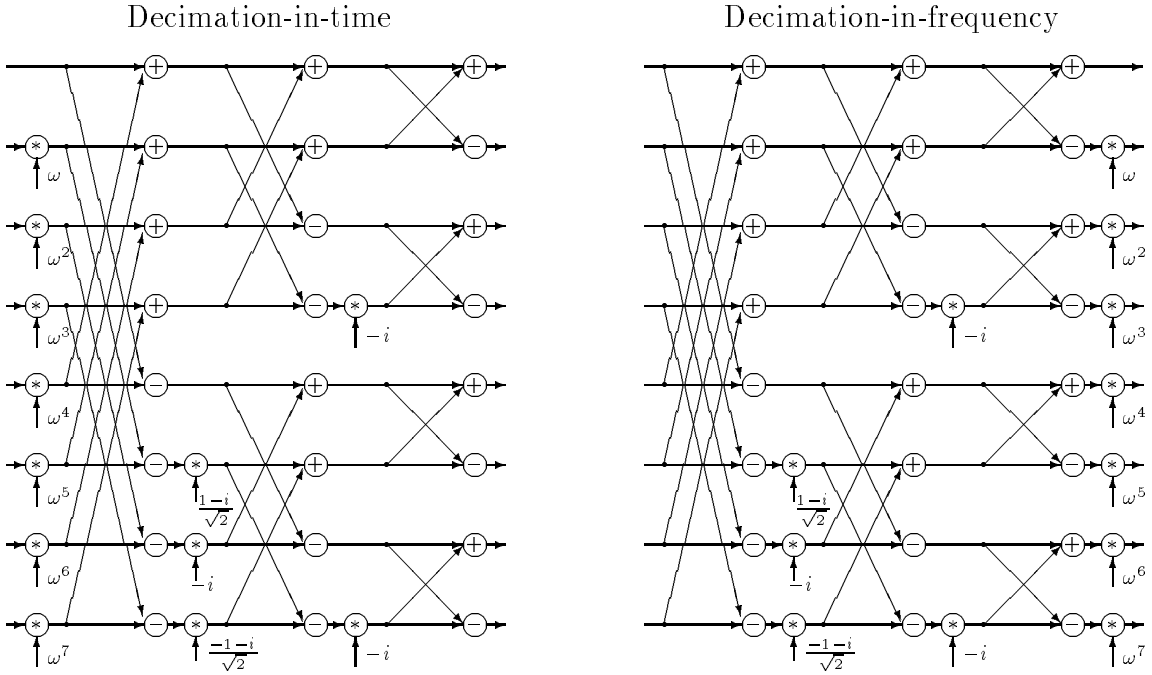

Figure 4: Radix-8 decimation-in-time and decimation-in-frequency kernels.

FFT	Arithmetic Operations			Storage References		
	Add	Mult	Total	Data	Twiddles	Total
Radix-2	$3Pp$	$2Pp$	$5Pp$	$4Pp$	Pp	$5Pp$
Radix-4	$\frac{22}{8}Pp$	$\frac{12}{8}Pp$	$\frac{17}{4}Pp$	$\frac{16}{8}Pp$	$\frac{6}{8}Pp$	$\frac{11}{4}Pp$
Radix-8	$\frac{66}{24}Pp$	$\frac{32}{24}Pp$	$\frac{49}{12}Pp$	$\frac{32}{24}Pp$	$\frac{14}{24}Pp$	$\frac{23}{12}Pp$

Table 1: Arithmetic and memory operations for radix-2, 4, and 8 FFTs.

register set in the floating-point processors of the Connection Machine systems, as discussed in Section 5.3. At the next level in the memory hierarchy, the local memory, the radix is equal to the size of the local data set.

The outline of the paper is as follows. We first briefly discuss the issues of the data allocation, or layout, among the memory modules. We then discuss the communication requirements of Cooley-Tukey FFT on multi-processors, specifically on Boolean cube configured processors. We compare the requirements of a direct pipelined algorithm, and algorithms based on bi-section, or multi-section, assuming concurrent communication on all channels of every processor, which is relevant for the Connection Machine systems. In Section 4 we discuss the computation and storage of twiddle factors for distributed FFT computations, and show how the storage requirements are related to the data layout, and the type of FFT being used. We then present results from our implementation on the Connection Machine system CM-200. All performance data are obtained for complex-to-complex FFT.

2 Data Allocation

In a distributed memory multi-processor architecture data is typically distributed uniformly across the memory modules at compile time, in order to maximize the potential concurrency in computation. If there are more data items than processors, then several data elements must be allocated to the same memory module. In a *consecutive* data allocation [7] successive elements are allocated to the same memory module. With n bits assigned to the encoding of processor addresses, the mapping of the array indices to machine addresses can be viewed as follows, where x_i denotes a bit in the encoding of the data indices:

Consecutive assignment:

$$(\underbrace{x_{p-1} \ldots x_{p-n}}_{rp} \underbrace{x_{p-n-1} x_{p-n-2} \ldots x_0}_{vp}).$$

The field denoted rp encodes *real processor* addresses as opposed to *memory* addresses, vp. In *cyclic* assignment the lowest order bits in the encoding of array indices are mapped to the processor address field.

Cyclic assignment:

$$(\underbrace{x_{p-1} x_{p-2} \ldots x_n}_{vp} \underbrace{x_{n-1} x_{n-2} \ldots x_0}_{rp}).$$

All data elements with the same n low order bits of their indices reside in the same processor. In the consecutive assignment the indices of all elements in a processor have the same n high order bits. The consecutive and cyclic allocations of a 32 element one-dimensional array among 8 processors are illustrated in Figure 5. We consider the impact of these forms of data allocation on the data motion requirements for the FFT.

For multi-dimensional arrays each axis is often encoded separately, as for instance is the case in the Connection Machine programming systems [21]. Still, there is an issue of how

Consecutive data allocation

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Cyclic data allocation

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Figure 5: Consecutive and cyclic data allocation of 32 elements to 8 processors.

to partition the processors among the axis of the data array. In the Connection Machine systems the configuration can be controlled through compiler directives. We discuss how to configure the processors for optimum performance in Section 5.

3 Communication requirements for the FFT

The data interaction in stage q of a radix-2 FFT is between data elements i and $i \oplus 2^{p-q-1}$, $i \in \{0, 1, \ldots, 2^p - 1\}$, where $q \in \{0, 1, \ldots, p-1\}$. The input stage is stage 0. The symbol $\oplus$ denotes the bit-wise exclusive-or function. Hence, in a radix-2 FFT the data interaction is between data elements that differ only in one bit in the encoding of their respective indices, starting with the most significant bit and progressing towards the least significant bit. For a radix-2^r FFT the interaction in stage s is between data that differ in bits $\{p-(s+1)r, p-(s+1)r+1, \ldots, p-sr-1\}$, where $s \in \{0, 1, \ldots, \frac{p}{r} - 1\}$ and we for simplicity assume that r divides p. For arbitrary p a collection of radices is needed.

Since the data interaction proceeds from the most significant digit in the encoding of the data indices towards the least significant digit, the first $\frac{n}{r}$ radix-2^r stages involve data motion between processors, when the data allocation is of the consecutive type. For the cyclic data allocation, the last $\frac{n}{r}$ stages involves communication. The data motion fits multiprocessors with processors interconnected as a Boolean cube network very well. Processors in such a network can be given addresses such that adjacent processors differ in precisely one bit, and conversely, there is an adjacent processor for every bit in the processor address. Hence, processors j and $j \oplus 2^m$ are adjacent for every $m \in \{0, 1, \ldots, n-1\}$, and any $j \in \{0, 1, \ldots, 2^n - 1\}$. Clearly, for a radix-2 FFT, stages corresponding to index bits mapped to the processor address field imply communication between directly connected processors. No other communication is required. A radix-2^r FFT requires communication between processors forming r-dimensional subcubes.

For an example of the communication needs consider Figure 5. It is easy to see that the first three stages of a radix-2 FFT with consecutive data allocation correspond to communication between directly connected processors. A radix-8 stage requires communication between all 8 processors. For the cyclic allocation, the last three stages require communication between directly connected processors. In a direct implementation of the splitting formulas for a

radix-2 FFT a pair of processors exchanges a pair of data elements, then one computes the "top", and one the "bottom". Each stage requires that $\frac{P}{N}$ elements be exchanged for a transform on P elements distributed evenly over N processors. There are n such stages for the axis subject to transformation distributed across $N = 2^n$ processors. In a multi-processor network with at least n channels per processor, such as in a Boolean n-cube, only one out of n communication channels are used.

A radix-2^r algorithm implemented in an analogous manner would require that each processor in r-dimensional subcubes sends one data element to every other processor in the subcube to which it belongs. After this all-to-all broadcast within r-cubes [1, 10] each processor computes one output value for the radix-2^r kernel. For each all-to-all broadcast r channels can be used concurrently. The number of element transfers in sequence for each all-to-all broadcast in r-cubes is $\frac{2^r-1}{r}$ [10]. The required temporary storage is $2^r - 1$. For large r the increased utilization of the communication system is accomplished at a significant expense in temporary storage.

The radix-2 implementation presented above suffers from a slight load imbalance in addition to the inefficiency in using the communication system. One of the processors in a pair computes a complex addition, while the other computes a complex multiplication and a complex subtraction. The radix-2^r algorithm yields a better load balance, but this gain is accomplished at the expense of redundant computations. We now consider a few alternative implementation strategies that yield both increased communication and computational efficiency. These alternatives were all considered for the Connection Machine implementation. The implementation is described in Section 5.

3.1 Direct pipelining

Pipelining the communications and computations for successive stages in the FFT is a straightforward way of increasing the utilization of the communication system. Pipelining allows d communication channels on every processor to be used concurrently in computing an FFT on an array axis distributed over 2^d processors of a Boolean cube network. The idea is illustrated for a radix-2 FFT in Figure 6. In the first communication data is exchanged in the most significant cube dimension. After the splitting formulas have been evaluated for these data items, they are ready for the second stage of the FFT. In the second communication the first memory locations in all processors are exchanged in the second most significant cube dimension, while the second memory locations are exchanged in the most significant cube dimension. From the third communication stage all communication channels are used in every exchange, until all local memory locations have been touched, at which point the shut-down of the communications pipeline starts.

The idea of pipelining the communications for the FFT computations can also be understood by observing that for a consecutive data allocation over 2^d processors, the first d stages can be viewed as $\frac{P}{N}$ distinct FFTs, each with one data point per processor. Each such FFT requires communication in the dimensions $d-1, d-2, \ldots, 1, 0$, one for each stage of the FFT. Hence, when the first stage of the first FFT is computed, dimension $d-1$ is free to be

Time Step	Memory location	Processor							
		0	1	2	3	4	5	6	7
0	0	2	2	2	2	2	2	2	2
	1	-	-	-	-	-	-	-	-
	2	-	-	-	-	-	-	-	-
	3	-	-	-	-	-	-	-	-
	4	-	-	-	-	-	-	-	-
1	0	1	1	1	1	1	1	1	1
	1	2	2	2	2	2	2	2	2
	2	-	-	-	-	-	-	-	-
	3	-	-	-	-	-	-	-	-
	4	-	-	-	-	-	-	-	-
2	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2
	3	-	-	-	-	-	-	-	-
	4	-	-	-	-	-	-	-	-
3	0	-	-	-	-	-	-	-	-
	1	0	0	0	0	0	0	0	0
	2	1	1	1	1	1	1	1	1
	3	2	2	2	2	2	2	2	2
	4	-	-	-	-	-	-	-	-

Figure 6: The first four steps of a direct pipelined radix-2 FFT.

used for the computations of the next FFT.

The radix-2, pipelined FFT requires $\frac{P}{N} + d - 1$ element transfers in sequence for an axis distributed over d cube dimensions, and with $\frac{P}{N}$ elements per processor. Note that for multi-dimensional FFTs, d is typically not equal to n, since more than one axis may be distributed over several processors. If $\frac{P}{N} >> d$, then the communication system is fully utilized effectively all the time. Pipelining offers an improvement in the communication efficiency by a factor of d over the naive implementation, for $\frac{P}{N} >> d$. It is easily verified that the claims are true for both the consecutive and cyclic data allocation. In the following we refer to the above algorithm as the "direct pipelined algorithm".

The idea of pipelining the communication and computations for successive FFT stages can be applied to radix-2^r FFT for $r > 1$, but the pipelined radix-2 FFT offers better overall efficiency for the reasons given in the previous section.

3.2 Bi-section

Even though the direct pipelined algorithm above uses the communication system to about 100%, the algorithm actually requires about twice the communication of implementations based on bi-section [14], or multi-section, or so called i-cycles [4, 20, 22]. The notion of i-cycles for the computation of FFTs as used in [20, 22] is equivalent to our notion of bi-section.

Proc. id	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
initial	0	1	2	3	4	5	6	7
alloc.	8	9	10	11	12	13	14	15
after	0	1	2	3	8	9	10	11
1st exch.	4	5	6	7	12	13	14	15
after	0	1	4	5	8	9	12	13
2nd exch.	2	3	6	7	10	11	14	15
after	0	2	4	6	8	10	12	14
3rd exch.	1	3	5	7	9	11	13	15

Figure 7: The data distribution for a radix-2 FFT based on bi-section with cyclic data allocation.

We focus on the use of the idea for communication systems with concurrent communication on multiple channels, whereas the development in [20, 22] assumes communication on one channel at a time. This difference in assumption of the capabilities of the communication system affects the utility of the idea in a fundamental way. The idea of using bi-section to achieve load balance and communication efficiency on Boolean cube networks is not new. It has been used previously for the solution of systems of tridiagonal equations [8].

The idea of computing an FFT through bi-section is illustrated in Figure 7 for a cyclic data allocation with two data elements per processor. The table shows the location of the data indices through the course of the algorithm. The first stage with the cyclic allocation requires no communication. Each processor evaluates one complete splitting formula. In the first exchange on the most significant processor dimension, the first half of the processors exchange the content of their second memory location with the content of the first memory location of the second half of the processors. After this exchange each processor can again perform the computations for one splitting formula, this time for the second stage of a radix-2 FFT. The exchange proceeds on successively lower processor dimensions, but use the same memory locations. Processors with the address bit 0 for the dimension subject to exchange, exchange their second memory location, while processors with the address bit 1 exchange their first memory location.

In each exchange a processor sends one out of two data elements identified by a local memory address bit. All processors evaluate $\frac{P}{2N}$ complete splitting formulas after each communication of $\frac{P}{2N}$ elements per processor. The load is perfectly balanced, and only half as much data is exchanged for each FFT stage. The idea of pipelining can be used in combination with the bi-section idea to fully utilize the communication system. Figure 8 shows the first few exchanges for a pipelined bi-section algorithm.

The factor of two gain in communication efficiency by using bi-section may not be fully realizable, or realizable at all for a consecutive data allocation. To see this fact we apply the bi-section idea to the consecutive allocation, as shown in Figure 9. Note that communication in the most significant dimension must be performed twice. With concurrent communication on all channels a pipelined bi-section algorithm requires $2\frac{P}{2N} + d - 1$ element transfers in

Time Step	Memory location	Processor							
		0	1	2	3	4	5	6	7
0	0	-	-	-	-	2	2	2	2
	1	2	2	2	2	-	-	-	-
	2	-	-	-	-	-	-	-	-
	3	-	-	-	-	-	-	-	-
	4	-	-	-	-	-	-	-	-
	5	-	-	-	-	-	-	-	-
1	0	-	-	1	1	-	-	1	1
	1	1	1	-	-	1	1	-	-
	2	-	-	-	-	2	2	2	2
	3	2	2	2	2	-	-	-	-
	4	-	-	-	-	-	-	-	-
	5	-	-	-	-	-	-	-	-
2	0	-	0	-	0	-	0	-	0
	1	0	-	0	-	0	-	0	-
	2	-	-	1	1	-	-	1	1
	3	1	1	-	-	1	1	-	-
	4	-	-	-	-	2	2	2	2
	5	2	2	2	2	-	-	-	-
3	0	-	-	-	-	-	-	-	-
	1	-	-	-	-	-	-	-	-
	2	-	0	-	0	-	0	-	0
	3	0	-	0	-	0	-	0	-
	4	-	-	1	1	-	-	1	1
	5	1	1	-	-	1	1	-	-

Figure 8: The first four steps of a pipelined bi-section based radix-2 FFT.

Proc. id	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
initial	0	2	4	6	8	10	12	14
alloc.	1	3	5	7	9	11	13	15
after	0	2	4	6	1	3	5	7
1st exch.	8	10	12	14	9	11	13	15
after	0	2	8	10	1	3	9	11
2nd exch.	4	6	12	14	5	7	13	15
after	0	4	8	12	1	5	9	13
3rd exch.	2	6	10	14	3	7	11	15
after	0	4	8	12	2	6	10	14
4th exch.	1	5	9	13	3	7	11	15

Figure 9: The data distribution for a radix-2 FFT based on bi-section with consecutive data allocation.

sequence, ignoring a possible overlap between the second exchange in the most significant dimension and the pipeline filling time. Hence, for the consecutive data allocation and concurrent communication on all channels, the communication requirements are the same as for the direct pipelined algorithm.

The FFT implementation based on bi-section reorders the data, in addition to the bit-reversal due to the FFT itself, unlike the direct pipelined FFT. That a reordering takes place is apparent from Figures 7 and 9, which both show the location of the original data indices. The data motion caused by the sequence of bi-sections implements an *unshuffle*. An unshuffle is the inverse of a shuffle, which perfectly interleaves the first and second half of a set of numbers. For instance, a shuffle on the set $\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$ produces the set $\{0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15\}$. Changing the last data ordering to the first corresponds to the reordering in Figure 7.

The data reordering for the cyclic data allocation can be represented formally in terms of the encoding of the address space as shown below. The overline marks address bits that have been exchanged in the step indicated on the left. For instance, after the first exchange step bits x_n and x_{n-1} have been exchanged. In an exchange, only data that differ in the values of the index bits are moved. For instance, in the first exchange only data for which $x_n \oplus x_{n-1} = 1$ are moved.

$$
\begin{array}{ll}
\text{Initial allocation:} & (\underbrace{x_{p-1} \ldots x_n}_{vp} \underbrace{x_{n-1} x_{n-2} \ldots x_0}_{rp}) \\
\text{Step 1:} & (\underbrace{x_{p-1} \ldots \overline{x_{n-1}}}_{vp} \underbrace{\overline{x_n} x_{n-2} \ldots x_0}_{rp}). \\
\text{Step 2:} & (\underbrace{x_{p-1} \ldots \overline{x_{n-2}}}_{vp} \underbrace{x_n \overline{x_{n-1}} x_{n-3} \ldots x_0}_{rp}) \\
\vdots & \vdots \\
\text{Step } n\text{:} & (\underbrace{x_{p-1} \ldots \overline{x_0}}_{vp} \underbrace{x_n \ldots \overline{x_1}}_{rp}).
\end{array}
$$

In this example the same memory dimension is used for all exchanges. Upon completion of the bi-section process the bits in the processor address together with the local memory bit used for all exchanges have been subject to a right cyclic shift, which defines an unshuffle. The local memory bit can be chosen arbitrarily. Using the least significant memory bit implies that every other location is subject to exchange, and the stride is two. If the most significant memory dimension is used, then a block equal to half of the local memory is exchanged, and the stride is one within the block. The number of memory references are the same, but architectural characteristics such as page faults, communications overhead, etc., may make the strategy for selecting the local memory bit important with respect to performance.

For the consecutive data allocation we use the exchange sequence for the cyclic allocation augmented with one additional exchange, as illustrated below

$$
\begin{array}{ll}
\text{Initial allocation:} & (\underbrace{x_{p-1}x_{p-2}\ldots x_{p-n}}_{rp}\underbrace{x_{p-n-1}x_{p-n-2}\ldots x_0}_{vp}) \\
\text{Step 1:} & (\underbrace{\overline{x_{p-n-1}}x_{p-2}\ldots x_{p-n}}_{rp}\underbrace{\overline{x_{p-1}}x_{p-n-2}\ldots x_0}_{vp}). \\
\text{Step 2:} & (\underbrace{x_{p-n-1}\overline{x_{p-1}}\ldots x_{p-n}}_{rp}\underbrace{\overline{x_{p-2}}x_{p-n-2}\ldots x_0}_{vp}) \\
\vdots & \vdots \\
\text{Step } n\text{:} & (\underbrace{x_{p-n-1}x_{p-1}\ldots\overline{x_{p-n+1}}}_{rp}\underbrace{\overline{x_{p-n}}x_{p-n-2}\ldots x_0}_{vp}) \\
\vdots & \vdots \\
\text{Step } n+1\text{:} & (\underbrace{\overline{x_{p-n}}x_{p-1}\ldots x_{p-n+1}}_{rp}\underbrace{\overline{x_{p-n-1}}x_{p-n-2}\ldots x_0}_{vp}).
\end{array}
$$

In this case an unshuffle permutation has been performed on the processor address field. The local memory address field is not reordered, as can be seen in Figure 9. Pairs of local memory locations contain even-odd pairs of successive data indices.

3.3 Multi-section

The idea of bi-section can be generalized to multi-section to support high radix FFT. A $R = 2^r$ way splitting implies matrix transposition, or *all-to-all personalized communication* [10, 9, 11] in r-dimensional subcubes. After each 2^r-sectioning step a radix-2^r FFT can be performed locally in each processor. Figure 10 illustrates multi-sectioning for the inter-processor communication steps for $p = 6$ and $n = 4$ and cyclic data allocation. The numbers in the table are the initial indices. The first partitioning step is a matrix transposition within each subcube of dimension two with respect to the two highest order real processor dimensions. For instance, processors 0, 4, 8 and 12 are in the same subcube. The bit interchanges corresponding to multi-sectioning is illustrated below for cyclic data allocation. As in the bi-section case there exist many ways in which partitioning of the local data can be performed throughout the algorithm, resulting in different final orderings.

$$
\begin{array}{ll}
\text{Initial allocation:} & (\underbrace{x_{p-1}x_{p-2}\ldots x_n}_{vp}\underbrace{x_{n-1}\ldots x_0}_{rp}). \\
\text{1st } 2^r\text{-section.} & (\underbrace{\overline{x_{n-1}x_{n-2}\ldots x_{n-r}}x_{p-r-1}x_{p-r-2}\ldots x_n}_{vp}\underbrace{\overline{x_{p-1}x_{p-2}\ldots x_{p-r}}x_{n-r-1}\ldots x_0}_{rp}). \\
\text{2nd } 2^r\text{-section} & (\underbrace{\overline{x_{n-r-1}x_{n-r-2}\ldots x_{n-2r}}x_{p-r-1}x_{p-r-2}\ldots x_n}_{vp} \\
 & \quad \underbrace{x_{p-1}x_{p-2}\ldots x_{p-r}\overline{x_{n-1}x_{n-2}\ldots x_{n-r}}x_{n-2r-1}x_{n-2r-2}\ldots x_0}_{rp}). \\
\vdots & \vdots \\
\text{Step } \frac{n}{r}\text{:} & (\underbrace{\overline{x_{r-1}x_{r-2}\ldots x_0}x_{p-r-1}x_{p-r-2}\ldots x_n}_{vp} \\
 & \quad \underbrace{x_{p-1}x_{p-2}\ldots x_{p-r}x_{n-1}x_{n-2}\ldots x_{2r}\overline{x_{2r-1}x_{2r-2}\ldots x_r}}_{rp}).
\end{array}
$$

Proc. id.	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initially	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51
1st	4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
part.	8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
	12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63
	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
2nd	1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
part.	2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
	3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63

Figure 10: The data distribution for a 4-sectioning, radix-4 FFT for 16 processors and 4 elements per processor.

For a 4-section algorithm, two bits are involved in every step. For a 2^r-section algorithm, r bits are involved in each permutation. Sectioning steps for successive radix-2^r stages involve consecutive blocks of r dimensions. The communication for the 2^r-sectioning on blocks of r different processor dimensions can be pipelined. The number of element transfers in sequence is $\frac{P}{2N} + (\lceil \frac{n}{r} \rceil - 1)2^{r-1}$ for cyclic data allocation. An in-place sectioning requires that $2^r \leq \frac{P}{N}$, or $r \leq p - n$, since the size of the local data set involved in a 2^r-section is 2^r. For $p \geq 2n$ ($P \geq N^2$) a 2^n-sectioning minimizes the number of element transfers in sequence, since there is no pipeline start-up or shut down in this case. Next to 2^n-sectioning, 4-sectioning is the best choice with respect to the number of element transfers in sequence. Bi-sectioning is insignificantly inferior with respect to the inter-processor communication requirements. For small values of r the variance in communication efficiency is small, and the choice of r is largely determined by the efficiency in evaluating the splitting formulas.

With a consecutive data allocation the r most significant processor dimensions must be used twice, and a pipelined multi-section algorithm requires $\frac{P}{N} + (\lceil \frac{n}{r} \rceil - 1)2^{r-1}$ element transfers in sequence, essentially the same as for the direct pipelined algorithm.

3.4 Discussion and summary of algorithms

In all derivations above the input order was normal. With the input in bit-reversed order the traversal of the address bits proceeds from the lowest to the highest order bit. With respect to the communication issues the roles of the consecutive and cyclic mapping are interchanged.

Forward and inverse transforms only differ in that one is computed using the conjugate values of the twiddle factors of the other. There are no particular issues with respect to multi-processors for one that is not present in the other.

Algorithm	Consecutive allocation		
	Element transfers	Input order	Output order
Direct Pipeline	$\frac{P}{N} + n - 1$	Normal	Bit-reverse
Bi-section	$\frac{P}{N} + n - 1$	Normal	Shuffled & Bit-reversed
Multi-section	$\frac{P}{N} + (\lceil \frac{n}{r} \rceil - 1)2^{r-1}$	Normal	Shuffled & Bit-reversed
	Cyclic allocation		
Direct Pipeline	$\frac{P}{N} + n - 1$	Normal	Bit-reverse
Bi-section	$\frac{P}{2N} + n - 1$	Normal	Shuffled & Bit-reverse
Multi-section	$\frac{P}{2N} + (\lceil \frac{n}{r} \rceil - 1)2^{r-1}$	Normal	Shuffled & Bit-reversed

Table 2: Communication requirements for unordered transforms with concurrent communication on all channels, and consecutive and cyclic ordering.

A multi-dimensional FFT can be performed as a sequence of one-dimensional FFTs for the different axes. Performed in this way the only issue unique to multi-dimensional FFT is how to partition the set of processors among the axes. We discuss this issue in the context of the Connection Machine implementation, see Section 5.

The communication requirements for the consecutive and cyclic data allocations are summarized in Table 2. The communication requirements assume concurrent communication on all channels. With a consecutive data allocation all algorithms yield the same communication requirements for an unordered transform, with data in normal input order. The output ordering is bit-reversed for the direct pipelined algorithm, and shuffled bit-reversed for the pipelined bi-section or multi-section algorithms. With a cyclic data allocation, and normal input order, the bi-section and multi-section type algorithms require approximately half as many element transfers in sequence as the direct pipelined algorithm. With a bit-reversed input order all algorithms essentially have the same communication requirements for the cyclic data allocation, while the bi-section or multi-section algorithms offer a reduction in the communications requirements for the consecutive data allocation.

For an ordered transform an explicit reordering phase is required. Interleaving the reordering with the FFT computation offers no gain in communications efficiency, when all communications channels are utilized for the unordered transform, unlike the case where only one channel at a time is used [20, 22]. The reordering requires the same number of element transfers in sequence for a pure bit-reversal operation, or a combined shuffle with bit-reversal, assuming concurrent communication on all channels [12, 11]. The number of element transfers in sequence for the reordering is $\frac{P}{2N}$. For details see [12, 11].

4 Twiddle Factors

The total number of twiddle factors needed for a radix-R FFT of size P is $(R-1)\frac{P}{R}$. For the computation of an FFT on a distributed memory architecture using precomputed twiddle factors, it is important to minimize the need for either redundant storage of twiddle factors or communication of twiddle factors should they be required in a processor different from the one in which they are stored.

In [15] we show that a radix-2 DIT FFT with precomputed twiddle factors, and data in normal order allocated consecutively, requires a maximum of $\frac{P}{2N} + d - 2$ twiddle factors in a processor. A DIF FFT with bit-reversed input order and consecutive allocation requires the same twiddle factors. A radix-2 DIT FFT on normal order input allocated cyclically, or a DIF FFT on bit-reversed data allocated cyclically requires a maximum of $(n-1)\frac{P}{N}$ twiddle factors in a processor [15]. Hence, the data allocation has a significant impact on the need for twiddle factor storage. Below we give algorithms for computation of twiddle factor indices based on memory addresses, for high radix FFT.

4.1 Decimation-in-frequency FFT

We first give a formula for the twiddle factor indices for a radix-2 *in-place* DIF FFT with normal order input, then generalize the formula to radix-R DIF FFT. For the first radix-2 stage the twiddle factor index is $(a_{p-1}) \times (a_{p-2}a_{p-2}\ldots a_0)$ for the data element in location $(a_{p-1}a_{p-2}\ldots a_0)$. The radix-2 twiddle factors can be derived from the following iterative formulation of the DIF FFT.

$$
\begin{aligned}
\tilde{x}_{-1}(a_{p-1},\ldots,a_0) &= x(a_{p-1},\ldots,a_0) \\
\tilde{x}_0(a_{p-1},\ldots,a_0) &= (\tilde{x}_{-1}(0,a_{p-2},\ldots,a_0) + \omega_2^{a_{p-1}}\,\tilde{x}_0(1,a_{p-2},\ldots,a_0))\,\omega_P^{\langle a_{p-2},\ldots,a_0\rangle a_{p-1}} \\
\tilde{x}_1(a_{p-1},\ldots,a_0) &= (\tilde{x}_0(a_{p-1},0,a_{p-3},\ldots,a_0) + \omega_2^{a_{p-2}}\,\tilde{x}_0(a_{p-1},1,a_{p-3},\ldots,a_0))\,\omega_{\frac{P}{2}}^{\langle a_{p-3},\ldots,a_0\rangle a_{p-2}} \\
&\vdots \\
\tilde{x}_q(a_{p-1},\ldots,a_0) &= (\tilde{x}_{q-1}(a_{p-1},\ldots,0_{p-q-1},\ldots,a_0) + \omega_2^{a_{p-q-1}}\,\tilde{x}_{q-1}(a_{p-1},\ldots,1_{p-q-1},\ldots,a_0))\,\omega_{\frac{P}{2^q}}^{\langle a_{p-q-2},\ldots,a_0\rangle a_{p-q-1}} \\
&\vdots \\
\tilde{x}_{p-1}(a_{p-1},\ldots,a_0) &= (\tilde{x}_{p-2}(a_{p-1},\ldots,a_1,0) + \omega_2^{a_0}\,\tilde{x}_{p-2}(a_{p-1},\ldots,a_1,1))\,\omega_2^{0a_0} \\
X(a_{p-1},\ldots,a_0) &= \tilde{x}_{p-1}(a_0,\ldots,a_{p-1})
\end{aligned}
$$

Note that, in the expression for $\tilde{x}_{p-1}$, the value of $\omega_2^{0a_0}$ is 1: no twiddle factors are needed in the last stage. For a radix-R *in-place* DIF FFT with normal order input we let $s \in [0, u-1]$, where $u = \log_R P = \frac{p}{r}$, and $(d_{u-1}d_{u-2}\ldots d_0)$ be the addresses in base R. Then,

$$
\tilde{x}_{-1}(d_{u-1},\ldots,d_0) = x(d_{u-1},\ldots,d_0)
$$

$$
\begin{aligned}
\tilde{x}_s(d_{u-1},\ldots,d_0) &= \omega_{\frac{P}{R^s}}^{(d_{u-s-2},\ldots,d_0)\widehat{d_{u-s-1}}} \sum_{j=0}^{R-1} \tilde{x}_{s-1}(d_{u-1},\ldots,d_{u-s},j,d_{u-s-2},\ldots,d_0)\,\omega_R^{\widehat{d_{u-s-1}}j} \\
\tilde{x}_{u-1}(d_{u-1},\ldots,d_0) &= \omega_{\frac{P}{R^{u-1}}}^{0\widehat{d_0}} \sum_{j=0}^{R-1} \tilde{x}_{u-2}(d_{u-1},\ldots,d_1,j)\,\omega_R^{\widehat{d_0}j} \\
X(d_{u-1},\ldots,d_0) &= \tilde{x}_{u-1}(\widehat{d_0},\ldots,\widehat{d_{u-1}})
\end{aligned}
$$

where the bit-reversed value of a digit d_i is $\widehat{d_i}$ (bit-reversal of the digit occurs because we want to keep the same ordering as with the radix-2 computation). As in the radix-2 case, no twiddle factors are needed in the last stage: the value of $\omega_{\frac{P}{R^{u-1}}}^{0\widehat{d_0}}$ is 1.

The twiddle factor index for data in location $(d_{u-1}d_{u-2}\ldots d_0)$ is $\widehat{d_{u-1}} \times (d_{u-2}d_{u-3}\ldots d_0)$ for the first stage. For the second radix-R stage the set of twiddle factor indices are $\widehat{d_{u-2}} \times (d_{u-3}d_{u-4}\ldots d_0)2^r$. In general, for a radix-R *in-place* DIF FFT on normal order input data, the twiddle factor index for the data in location $(d_{u-1}d_{u-2}\ldots d_0)$ after the sth radix R stage is $\widehat{d_{u-s-1}} \times (d_{u-s-2}d_{u-s-3}\ldots d_0)2^{sr}$.

For a distributed data set we consider the need for twiddle factors in a processor first for the local stages, then for stages requiring communication. With a data set of size $P = 2^p$ in normal order distributed cyclically over $N = 2^n$ processors the computations corresponding to the first $\frac{u-n}{r}$ radix-2^r stages, are local to the processors. For simplicity, we assume that u and n are multiples of r. The twiddle factor indices for stage s required in processor $(d_{\frac{n}{r}-1}\ldots d_0)$ are $\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2}d_{u-s-3}\ldots d_{\frac{n}{r}}\}d_{\frac{n}{r}-1}\ldots d_0)2^{sr}$. The notation $\{\ldots\ldots\}$ denotes the set of all values that can be assumed by the digit string within the braces. When $\frac{P}{N}$ is a multiple of R, then $(\frac{P}{N}-1)$ twiddle factors are needed for the local stages.

The stages requiring communication correspond to computing $\frac{P}{N}$ independent FFTs of size N, each with one element per processor. All $\frac{P}{N}$ FFTs require the same set of twiddle factors in a processor. A total of at most $(\lceil\frac{n}{r}\rceil-1)(2^r-1)$ twiddle factors are needed in a processor for these stages (one set for each radix-R butterfly stage, except the last stage). The property that the twiddle factor only depends upon the processor address, and is the same for all local elements, is the same for the direct pipelined algorithm, and the bi-section or multi-section algorithms.

To summarize, the maximum number of distinct twiddle factors needed in a processor is $\frac{P}{N}+(\lceil\frac{n}{r}\rceil-1)(2^r-1)-1$ for cyclic data allocation, normal input order, and a radix-2^r DIF FFT of size P computed on N processors, $N \leq P$. Allocating twiddle factor storage uniformly across all processors yield a total twiddle factor storage of $P-N+(\lceil\frac{n}{r}\rceil-1)(2^r-1)N$, which for $P \gg N$ is about twice the storage required on a shared memory computer. The same twiddle storage is required for a bit-reversed input order, and a consecutive data allocation. Normal input order and consecutive data allocation, or cyclic allocation with bit-reversed input order would require considerably more storage, for the same reasons as in the radix-2 case [15].

4.2 Decimation-in-time FFT

As in the DIF case we first consider the radix-2 case, then generalize to the radix-2^r case.

$$
\begin{aligned}
\tilde{x}_{-1}(a_{p-1},\ldots,a_0) &= x(a_{p-1},\ldots,a_0) \\
\tilde{x}_0(a_{p-1},\ldots,a_0) &= \tilde{x}_{-1}(0,a_{p-2},\ldots,a_0) + \omega_2^{a_{p-1}}\,\omega_2^0\,\tilde{x}_{-1}(1,a_{p-2},\ldots,a_0) \\
\tilde{x}_2(a_{p-1},\ldots,a_0) &= \tilde{x}_1(a_{p-1},0,a_{p-3},\ldots,a_0) + \omega_2^{a_{p-2}}\,\omega_4^{\langle a_{p-1}\rangle}\,\tilde{x}_1(a_{p-1},1,a_{p-3},\ldots,a_0) \\
&\vdots \\
\tilde{x}_s(a_{p-1},\ldots,a_0) &= \tilde{x}_{s-1}(a_{p-1},\ldots,0_{p-s-1},\ldots,a_0) + \omega_2^{a_{p-s-1}}\,\omega_{2^{s+1}}^{\langle a_{p-s},\ldots,a_{p-1}\rangle}\,\tilde{x}_{s-1}(a_{p-1},\ldots,1_{p-s-1},\ldots,a_0) \\
&\vdots \\
\tilde{x}_{p-1}(a_{p-1},\ldots,a_0) &= \tilde{x}_{p-2}(a_{p-1},\ldots,a_1,0) + \omega_2^{a_0}\,\omega_P^{\langle a_1,\ldots,a_{p-1}\rangle}\,\tilde{x}_{p-2}(a_{p-1},\ldots,a_1,1) \\
X(a_{p-1},\ldots,a_0) &= \tilde{x}_{p-1}(a_0,\ldots,a_{p-1})
\end{aligned}
$$

Note that, in the expression for $\tilde{x}_0$, the value of ω_2^0 is 1: no twiddle factors are needed in the first stage. A radix-R, in-place, DIT FFT with input data in normal order can be written as

$$
\begin{aligned}
\tilde{x}_{-1}(d_{u-1},\ldots,d_0) &= x(d_{u-1},\ldots,d_0) \\
\tilde{x}_s(d_{u-1},\ldots,u_0) &= \sum_{j=0}^{R-1} \omega_R^{\widehat{d_{u-s-1}}\,j}\,\omega_{R^{s+1}}^{\langle\widehat{d_{u-s}},\ldots,\widehat{d_{u-1}}\rangle j}\,\tilde{x}_{s-1}(d_{u-1},\ldots,d_{u-s},j,d_{u-s-2},\ldots,d_0) \\
\tilde{x}_{u-1}(d_{u-1},\ldots,u_0) &= \sum_{j=0}^{R-1} \omega_R^{\widehat{d_0}\,j}\,\omega_P^{\langle\widehat{d_1},\ldots,\widehat{d_{u-1}}\rangle j}\,\tilde{x}_{u-2}(d_{u-1},\ldots,d_1,j) \\
X(d_{u-1},\ldots,d_0) &= \tilde{x}_{u-1}(\widehat{d_0},\ldots,\widehat{d_{u-1}})
\end{aligned}
$$

The indices of the twiddle factors are all zero for the first stage, $j \times \widehat{d_{u-1}}\,2^{p-2r}$ for the second radix-R stage, and $j \times (\widehat{d_{u-s}}\ldots\widehat{d_{u-1}})2^{p-(s+1)r}$ for stage number s. Note, that the address is bit-reversed and shifted for the proper exponent. If the P complex data points are allocated consecutively and are in normal order, then the data in address location $(d_{u-1}d_{u-2}\ldots d_0)$ requires twiddle factors with indices $\{j\} \times (\{\widehat{d_{u-s}}\ldots d_{u-\frac{n}{r}-1}\}\widehat{d_{u-\frac{n}{r}}}\ldots\widehat{d_{u-1}})2^{p-(s+1)r}$ for stage s of an in-place DIT algorithm. With a consecutive data allocation the processor address bits form the high order bits of the element index. The first $\frac{n}{r}$ radix-R butterfly stages correspond to $\frac{P}{N}$ independent FFTs of size N. All these FFTs require the same set of twiddle factors. The local addresses do not enter into the index computation. Moreover, the first stage does not require any twiddle factor. The last $u - \frac{n}{r}$ radix-R stages are local to a processor. The maximum total number of twiddle factors required in a processor is $\frac{P}{N} + (\lceil\frac{n}{r}\rceil - 1)(2^r - 1) - 1$, the same as for cyclic data allocation, normal input order, and in-place DIF FFT. The set of twiddle factors required in a processor is identical to those required for consecutive data allocation, bit-reversed input order and a DIF, in-place FFT. The number of twiddle factors required for a DIT FFT with input data in bit-reversed order and a consecutive data allocation is excessive, see [15].

FFT	Data alloc.	Twiddle index stage s	Max. number of twiddles per proc.
Normal input order			
DIT	consec.	$\{j\} \times (\{\widehat{d_{u-s} \ldots d_{u-\frac{n}{r}-1}}\}\widehat{d_{u-\frac{n}{r}} \ldots d_{u-1}})2^{p-(s+1)r}$	$\frac{P}{N} + \frac{n}{r} - 2$
DIF	cyclic.	$\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2}d_{u-s-3} \ldots d_{\frac{n}{r}}\}d_{\frac{n}{r}-1} \ldots d_0)2^{sr}$	$\frac{P}{N} + \frac{n}{r} - 2$
Bit-reversed input order			
DIT	cyclic.	$\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2}d_{u-s-3} \ldots d_{\frac{n}{r}}\}d_{\frac{n}{r}-1} \ldots d_0)2^{sr}$	$\frac{P}{N} + \frac{n}{r} - 2$
DIF	consec.	$\{j\} \times (\{\widehat{d_{u-s} \ldots d_{u-\frac{n}{r}-1}}\}\widehat{d_{u-\frac{n}{r}} \ldots d_{u-1}})2^{p-(s+1)r}$	$\frac{P}{N} + \frac{n}{r} - 2$

Table 3: Radix-2^r twiddle factor storage as a function of input order.

4.3 Summary of twiddle factor storage requirements

The preferred combinations of data allocation and FFT type is summarized in Table 3.

For multi-dimensional FFT each axis has its set of twiddle factors. The twiddle factors for an axis is a subset of the twiddle factors for the longest axis. With axes of length $P_1 \times P_2 \times \ldots P_k$ the minimum number of twiddle factors is $\max_\ell (R-1)\frac{P_\ell}{R}$. With separate storage of the twiddle factors for each axis the total storage is $\sum_\ell (R-1)\frac{P_\ell}{R}$, which is still less than the storage required for a one-dimensional FFT of size $\Pi_\ell P_\ell$.

The Inverse Discrete Fourier Transform can be computed as a Discrete Fourier Transform by using conjugate twiddle factors.

5 A Connection Machine implementation.

5.1 Overview

The consecutive data allocation is used by all compilers for the Connection Machine systems. In our implementation a DIT FFT is used for data in normal input order, and a DIF FFT is used for bit-reversed input order. This combination of data input order and FFT type minimizes the requirements for twiddle factor storage. The inverse Discrete Fourier Transform is computed using conjugate twiddle factors. Multi-dimensional FFT are computed as a sequence of one-dimensional FFT, with all one-dimensional FFTs along an axis computed concurrently. For ease of implementation twiddle factors are allocated independently for each axis. For P data points allocated evenly over N processors the number of twiddle factors per processor for an axis allocated over 2^d processors is $\frac{P}{N} + d - 1$.

For simplicity and efficiency the direct pipelined algorithm is used for FFT stages requiring communication. Since all Connection Machine compilers use the consecutive data allocation scheme, and communication can be performed concurrently on all communication channels of

every processor, the bi-section and multi-section techniques do not offer any reduction in the communication time compared to the direct pipelined algorithm. Indeed, on the Connection Machine systems the bi-section and multi-section techniques require more time for a data exchange between a pair of processors than the direct pipelined algorithm. The reason for this difference is that the exchanges in the direct pipelined algorithm take place between memory locations with the same local addresses in different processors, while the other algorithms require that the elements in an exchange has different local memory addresses. Depending on algorithm [3, 11] the increase in the time for an exchange is in the range 30 - 100% for the Connection Machine systems CM-2 and CM-200. The increased communication time due to this reduction in communication efficiency is in many cases greater than the reduction in time for evaluating the splitting formulas by radix-4 or 8 kernels instead of by radix-2 kernels.

In the direct pipelined algorithm the FFT stages requiring communication are computed using a radix-2 algorithm. Local stages are computed using a mix of radix-2, 4 and 8 kernels. For efficiency, as many stages as possible are performed using the radix-8 kernels. To increase the efficiency of the radix-2 kernels for the inter-processor communication stages, data caching is performed as explained in Section 5.2.

Reordering for ordered transforms is performed explicitly. Interleaving the reordering with the computation of the unordered transform would not gain any efficiency with respect to communication, since all communication channels are already used by the unordered FFT. For details of algorithms see [3, 10, 11]. Timings are presented for both the unordered and ordered FFT.

All performance data presented below refer to complex-to-complex transforms performed on the Connection Machine system CM-200. The data is assumed to be presented in normal order for the DIT FFT, and bit-reversed order for the DIF FFT. A standard binary encoding of the indices for each axis is assumed. For Boolean cube multi-processors a common encoding of array indices is the binary-reflected Gray code encoding [19]. This encoding is also supported on the Connection Machine systems. FFT algorithms for this type of data encoding can be found in [13]. Those algorithms have not yet been implemented on the Connection Machine systems.

5.2 Organization of the inter-processor communication stages

For the inter-processor communication stages, direct pipelined radix-2 DIT or DIF FFT algorithms are used for normal and bit-reversed input order, respectively. For each inter-processor communication FFT stage, a single twiddle factor is needed for all local data elements. The total number of twiddle factors needed in each processor is equal to the number of processor dimensions d over which the data set is distributed. d is the number of FFT stages requiring communication. For the direct pipelined algorithm d data elements are exchanged in each communication, except during the start-up and shut down of the communications pipeline. d butterfly computations can be performed after each communication. The butterfly computations belong to different stages, and require different twiddles.

As is apparent from Figure 6 in Section 3.1 each local data element is updated d times in succession. No further updates are required for the inter-processor communication phase. In order to reduce the number of loads and stores to local memory, the local data items are cached in the register set of the floating-point unit. Twiddle factors are (re)read from memory. The data caching scheme is used for up to 10 dimensions. For 11 dimensions there are insufficient registers in the floating-point unit, resulting in a performance loss, as can be seen from the timings in Section 5.5.

Another detail that deserves to be mentioned addresses the SIMD (Single Instruction Multiple Data) nature of the Connection Machine systems CM-2 and CM-200. In the butterfly computations one processor in a pair performs a complex addition and the other a complex subtraction. By integrating the negation of one of the operands into the communication both arithmetic operations can be performed concurrently with no measurable loss in efficiency.

5.3 Organization of the local FFT stages.

The current floating-point unit of the Connection Machine has a register file of 32 registers, which is sufficient to keep all the twiddle factors and the temporary variables for the radix-2 and radix-4 kernels. For the radix-8 kernel, the twiddle factors are brought in from memory as they are needed, and only the temporary variables are kept in the registers. For kernels of higher radices, temporary results would have to be stored in memory. For that reason, we only implemented the radix-2, radix-4 and radix-8 kernels.

To handle data sets of any power-of-two (on-processor) size, it is necessary to mix kernels of different radices. Our implementation does as much of the computation as possible with radix-8 kernels, using one stage of radix-2 or radix-4 kernels to handle the remainder of the computation when the size is not a power of 8.

An FFT algorithm is typically expressed in terms of three nested loops. The outermost loop ranges over the stages of the FFT ("stage loop"). The two inner loops range over the groups (a group is a set of kernels which use the same set of twiddle factors) in each stage ("group loop") and over all the kernels in each group ("kernel loop"). With this organization, the twiddle factors for all the kernels in each group can be kept in the register file during the extent of the kernel loop; they are loaded at the beginning of the kernel loop and need not be loaded for each kernel. For radix-8 kernels, only part of the twiddle factors can be kept in registers, due to the register file size.

The number of groups and the number of kernels in a group change from stage to stage. The product of the number of groups and the number of kernels in a group is the total number of kernels in a stage, which is equal to the local FFT size divided by the size of the current kernel. Table 4 gives the number of groups and the number of kernels of size $R = 2^r$ per group, for a given radix-2^r butterfly stage s, when there are $\frac{P}{N}$ elements per processor. The loop structure is given by the following pseudocode (it assumes for simplicity that the kernel size is always the same, but it can be easily generalized):

`for` s`:=0 to` $\frac{p-n}{r} - 1$

FFT type	radix-2^r stage	nb_groups(s,r)	nb_kernels(s,r)
DIT, normal order input	s	2^{sr}	$\frac{P}{2^{(s+1)r}N}$
DIF, bit-rev. input	s	$\frac{P}{2^{(s+1)r}N}$	2^{sr}

Table 4: Number of groups and kernels per group.

```
for g:=0 to nb_groups(s,r)
  for k:=0 to nb_kernels(s,r)
    call kernel of size 2^r on the appropriate data
```

For example, an FFT of size 128 computed by DIF, consists of 3 stages:

- one stage of 16 groups, each with 1 radix-8 kernel
- one stage of 2 groups, each with 8 radix-8 kernels
- one stage of 1 group, with 64 radix-2 kernels

In the last stage, only the first radix-2 kernel needs to load the twiddle factor from memory to the register file; the other 63 kernels will use the twiddle factor already in the register file.

5.4 The Twiddle Factors

In stage s (as defined above), a radix-R FFT ($R = 2^r$) needs $R-1$ twiddle factors per kernel. Since all the kernels in one group use the same set of twiddle factors, the number of twiddle factors used in one stage is the number of groups multiplied by $R-1$. Hence, in stage s, with $\frac{P}{N}$ elements per processor, the DIT FFT needs $(R-1)2^{sr}$ different twiddle factors, and the DIF FFT needs $\frac{(R-1)}{2^{(s+1)r}}\frac{P}{N}$ twiddle factors. For all the stages, both of these add up to $\frac{P}{N}-1$. There are $P-N$ twiddle factors used in total for the local part of the FFT.

The DIT FFT is performed on data stored in normal order. For stage s, processor N_i needs the twiddle factors

$$\omega_{2^{n+(s+1)r}}^{j\times\widehat{N_i\|g}}, \quad j \in [1, R-1],$$

for the kernels in group g, where $x\|y$ is the concatenation of x and y. For the DIF FFT with cyclic data allocation and the input in bit-reversed order, the twiddle factors used by the kernels in group g in processor N_i at stage s are

$$\omega_{2^{p-sr}}^{j\times\widehat{N_i\|g}}, \quad j \in [1, R-1].$$

If the substitution $s \leftarrow \frac{p-n}{r} - s - 1$ is made in the expression above, we get exactly the expression for DIT twiddle factors. The DIF and the DIT FFT's are thus using the same set of twiddle factors, but are using them in reverse orders. Our implementation only uses one table of twiddle factors for both the DIT and the DIF FFT.

The following pseudo-code reflects exactly how the table of twiddle factors is stored in the processor memory. It generates them in the order used by the DIT FFT.

```
twiddle_pointer:=0
for s:=0 to (p-n)/r - 1
   for g:=0 to nb_groups(s,r)
      for j:=1 to 2^r - 1
         twiddle[twiddle_pointer] := ω^(j×N_i||g / 2^(n+(s+1)r))
         twiddle_pointer := twiddle_pointer + 1
```

5.5 Performance measurements

The performance measurements below have been made on a Connection Machine system CM-200 with 2048 64-bit floating-point units. All performance data refer to a complex-to-complex FFT, CCFFT, implemented as described above, and included as part of the Connection Machine Scientific Software Library version 3.0. Data is provided for both ordered and unordered FFT.

Performance of local FFTs for different array sizes is given in Table 5 and Figure 11. The peaks in Figure 11 correspond to array sizes for which only radix-8 kernels are used. The performance for 64-bit precision is about 75-80% of the performance for 32-bit precision. The difference is due to the fact that the data path between each floating-point unit and its memory is 32-bits wide. Data paths internal to the floating-point unit are 64-bits wide. The performance of the DIT kernels is 90 - 95% of the DIF kernel performance for most sizes. The difference is due to minor differences in the construction of arithmetic pipelines for the floating-point processor. Table 6 gives performance data for ordered local transforms. Large ordered transforms are about 10% slower than unordered transforms. For transforms of size 1024 the ordered transform is about 20% slower than the unordered transform. The ordering phase requires one traversal of memory regardless of the size of the array, whereas the computation of the FFT requires several traversals.

Timings for two- and three-dimensional CCFFT are given in Table 7, and shown in Figure 12. The significant increase in performance for the two-dimensional CCFFT between the 1024×1024 array and the 2048×2048 array is due to one of the axis being local to a processor for the larger array (there are 2048 processors). The subsequent minor decrease in performance for the next larger array is due to the fact that the axis distributed over all processors also has a local component of length two. This part of the axis requires a radix-2 kernel, which is less efficient than the radix-4, and the radix-8 kernels normally used. For reference, performance data for ordered two and three-dimensional transforms are given in Table 8. The execution time increases by 50 - 100% for our examples, considerably more than for entirely local transforms.

Optimal efficiency is attained by maximizing the number of axes that have no non-local component. Recall that with the pipelining of communications, the number of element transfers in sequence is $\frac{P}{N} + d - 1$, where $\frac{P}{N}$ is the number of elements per processor, and d the number of inter-processor dimensions over which an axis subject to transformation

Axis	Time, msec				Gflops/s			
	32-bit prec.		64-bit prec.		32-bit prec.		64-bit prec.	
length	DIT	DIF	DIT	DIF	DIT	DIF	DIT	DIF
32	0.18	0.17	0.23	0.21	9.077	9.476	7.211	7.896
64	0.35	0.33	0.43	0.39	11.293	12.032	9.170	10.171
128	0.84	0.80	1.12	1.07	10.958	11.396	8.157	8.567
256	1.94	1.82	2.48	2.24	10.825	11.504	8.449	9.378
512	3.92	3.66	4.92	4.42	12.051	12.887	9.599	10.669
1024	9.07	8.69	12.05	11.34	11.565	12.065	8.700	9.249
2048	20.66	19.44	26.60	24.01	11.167	11.865	8.671	9.609
4096	41.79	39.29	53.00	47.75	12.043	12.809	9.496	10.541
8192	93.65	89.69	123.92	115.60	11.644	12.159	8.800	9.434
16384	207.86	196.21	268.28	242.17	11.300	11.971	8.755	9.699
32768	419.82	395.98	535.17	482.54	11.989	12.711	9.405	10.431
65536	920.42	881.03	1213.82	1126.12	11.666	12.187	8.846	9.535
131072	2005.73	1897.08	2737.99	2593.42	11.376	12.027	8.333	8.798
262144	4355.31	4167.26			11.094	11.595		

Table 5: Performance data for local, unordered, CCFFT on a 2048 processor CM-200.

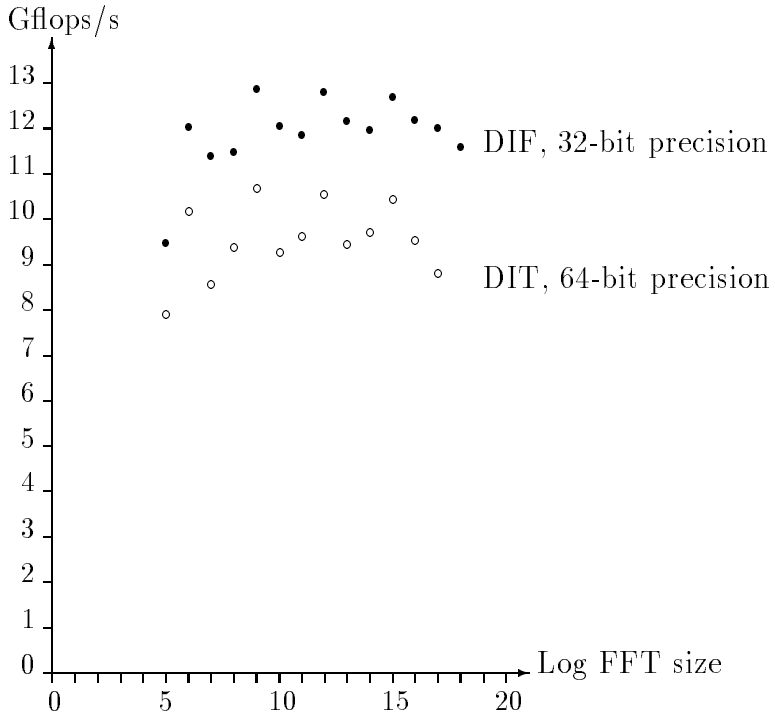

Figure 11: The performance of local, unordered, DIF CCFFT on a 2048 processor CM-200.

Axis	Time, msec				Gflops/s			
	32-bit prec.		64-bit prec.		32-bit prec.		64-bit prec.	
length	DIT	DIF	DIT	DIF	DIT	DIF	DIT	DIF
32	0.23	0.23	0.33	0.32	7.062	7.155	4.965	5.201
64	0.45	0.43	0.63	0.59	8.680	9.123	6.192	6.631
128	1.04	1.01	1.53	1.49	8.814	9.075	5.977	6.178
256	2.35	2.24	3.32	3.08	8.916	9.354	6.309	6.805
512	4.74	4.49	6.59	6.10	9.961	10.516	7.162	7.742
1024	10.74	10.37	15.45	14.74	9.765	10.116	6.786	7.112
2048	24.04	22.84	33.47	30.87	9.595	10.101	6.892	7.473
4096	48.65	46.16	66.83	61.58	10.346	10.903	7.531	8.173
8192	107.41	103.48	151.64	143.33	10.152	10.538	7.191	7.608
16384	235.51	223.91	323.89	297.81	9.973	10.490	7.252	7.887
32768	475.21	451.45	646.46	593.88	10.591	11.149	7.786	8.475
65536	1031.33	992.09	1436.59	1349.00	10.411	10.823	7.474	7.960
131072	2227.67	2119.29	3183.99	3039.30	10.243	10.766	7.166	7.507
262144	4801.31	4615.54			10.064	10.469		

Table 6: Performance data for local, ordered, CCFFT on a 2048 processor CM-200.

Axis	Time, msec				Gflops/s			
	32-bit prec.		64-bit prec.		32-bit prec.		64-bit prec.	
length	DIT	DIF	DIT	DIF	DIT	DIF	DIT	DIF
256×256	2.8	2.8	4.7	4.7	1.862	1.856	1.118	1.115
512×512	10.6	10.8	17.6	17.8	2.227	2.181	1.340	1.325
1024×1024	43.0	43.0	71.1	73.1	2.439	2.439	1.476	1.435
2048×2048	103.3	106.7	171.5	173.8	4.464	4.326	2.691	2.655
4096×4096	487.1	501.5	760.6	770.6	4.133	4.014	2.647	2.613
8192×8192	1868.1	1921.8	2986.0	3022.7	4.670	4.539	2.922	2.886
16384×16384	7470.4	7648.2	11846.3	11916.6	5.031	4.914	3.172	3.154
$64 \times 64 \times 64$	10.6	10.6	17.6	17.6	2.221	2.228	1.342	1.342
$128 \times 128 \times 128$	79.3	78.9	134.0	133.7	2.777	2.791	1.643	1.647
$256 \times 256 \times 256$	724.2	721.8	1180.4	1176.4	2.780	2.789	1.706	1.711
$512 \times 512 \times 512$	5608.4	5554.7	9227.5	9128.8	3.231	3.262	1.964	1.985

Table 7: Performance data for two and three-dimensional, unordered, CCFFT on a 2048 processor CM-200.

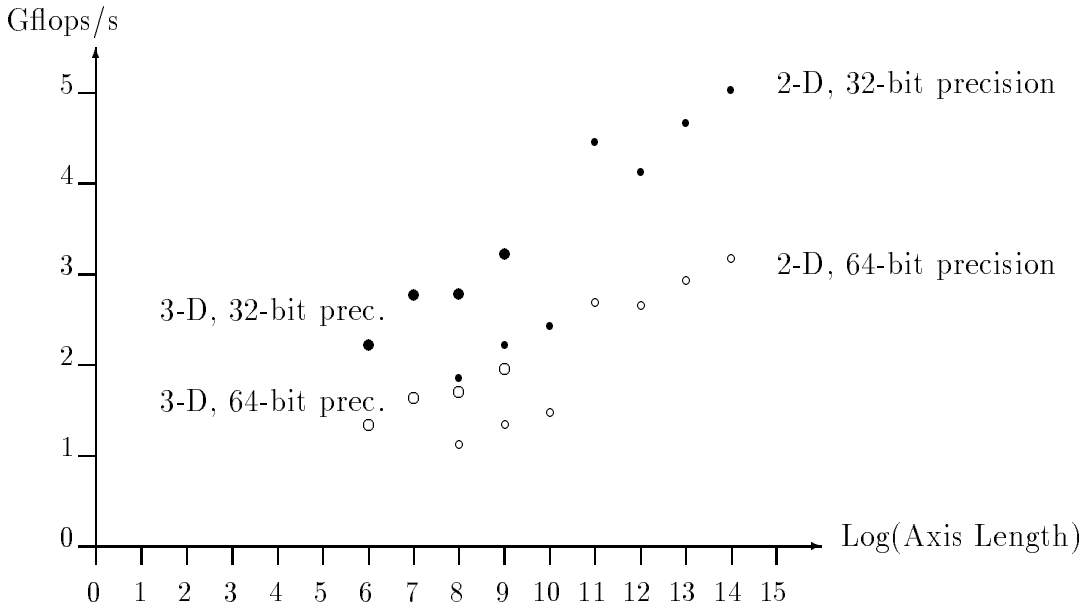

Figure 12: The execution rate for two and three-dimensional, unordered, DIT CCFFT on a 2048 processor CM–200.

Axis	Time, msec				Gflops/s			
	32-bit prec.		64-bit prec.		32-bit prec.		64-bit prec.	
length	DIT	DIF	DIT	DIF	DIT	DIF	DIT	DIF
256×256	4.62	4.63	8.05	8.06	1.134	1.132	0.652	0.650
512×512	16.77	16.98	29.66	29.77	1.407	1.389	0.795	0.793
1024×1024	68.55	68.71	122.73	124.39	1.530	1.526	0.854	0.843
2048×2048	183.36	186.66	329.43	331.76	2.516	2.472	1.401	1.391
4096×4096	907.03	923.01	1598.08	1609.40	2.220	2.181	1.260	1.251
8192×8192	3393.68	3448.63	6049.18	6091.06	2.571	2.530	1.442	1.432
16384×16384	13715.35	13894.09	24409.33	24475.00	2.740	2.705	1.540	1.535
$64 \times 64 \times 64$	18.89	18.89	32.10	32.11	1.249	1.249	0.735	0.735
$128 \times 128 \times 128$	123.10	122.71	221.53	221.27	1.789	1.794	0.994	0.995
$256 \times 256 \times 256$	1101.56	1100.11	1930.29	1927.64	1.828	1.830	1.043	1.044
$512 \times 512 \times 512$	8302.94	8251.76	14655.03	14560.63	2.182	2.196	1.236	1.244

Table 8: Performance data for two and three-dimensional, ordered, CCFFT on a 2048 processor CM-200.

Local axis length	Time, msec				Gflops/s			
	32-bit prec.		64-bit prec.		32-bit prec.		64-bit prec.	
	DIT	DIF	DIT	DIF	DIT	DIF	DIT	DIF
4096	485.49	500.34	757.98	770.68	4.147	4.024	2.656	2.612
2048	681.96	675.88	1132.61	1123.06	2.952	2.979	1.778	1.793
1024	654.05	649.90	1097.09	1091.76	3.078	3.098	1.835	1.844
512	653.05	648.55	1097.75	1092.43	3.083	3.104	1.834	1.843
256	654.92	649.30	1099.37	1089.70	3.074	3.101	1.831	1.848
128	645.21	641.13	1087.37	1082.66	3.120	3.140	1.852	1.860
64	645.33	641.32	1087.34	1082.70	3.120	3.139	1.852	1.859
32	654.82	649.23	1099.20	1089.65	3.075	3.101	1.832	1.848
16	652.59	647.95	1094.04	1088.97	3.085	3.107	1.840	1.849
8	654.74	650.95	1096.36	1093.12	3.075	3.093	1.836	1.842
4	679.62	674.27	1130.14	1123.14	2.962	2.986	1.781	1.793
2	487.10	501.52	760.58	770.56	4.133	4.014	2.647	2.613

Table 9: Performance of a two-dimensional unordered CCFFT on a 4096×4096 array computed on a 2048 processor CM-200.

is spread. The number of element transfers in sequence is approximately independent of the number of axis d, except if $d = 0$ in which case no communication is required. The performance variation once an axis is distributed across processors is minor, as can be seen in Table 9. For a two-dimensional FFT of shape 4096×4096 the worst performance once an axis is distributed across processors is at most 5% below the peak in 32-bit precision, and at most 3.5% below peak in 64-bit precision. The difference between a distributed axis, and a local axis is about 20% in 32-bit precision and close to 30% in 64-bit precision.

6 Summary and Discussion

We have shown that for consecutive data allocation, normal order input, and a Boolean cube interconnection network allowing concurrent communication on all channels of every processor, a direct pipelined radix-2 FFT and an FFT based on multi-section or i-cycles [20, 22] all yield essentially the same communication requirements. The number of element transfers in sequence is $\frac{P}{N} + d - 1$ for a transform on an array of size P distributed evenly over N processors, with the axis subject to transformation distributed over 2^d processors. For a cyclic data allocation and normal input order, or bit-reversed input order and consecutive data allocation, an FFT based on multi-section requires about half as many element transfers in sequence as a direct pipelined FFT.

We have also shown that with precomputed twiddle factors a decimation-in-time FFT for consecutive data allocation and normal order input, requires approximately the same total storage on a distributed memory architecture as on a shared memory architecture. No computation, or communication of twiddle factors is necessary with this amount of storage.

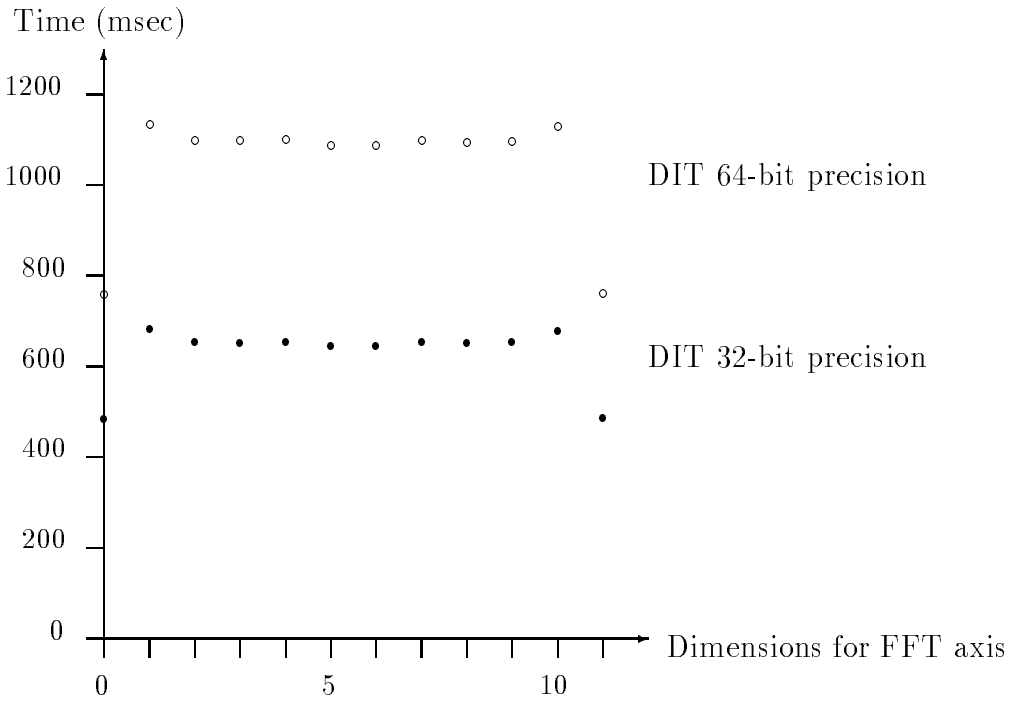

Figure 13: Total execution time for a two-dimensional unordered CCFFT on a 4096×4096 array as a function of the configuration of 2048 CM-200 processors.

A decimation-in-frequency FFT requires the same twiddle factors in the same processors if the input is in bit-reversed order and the data allocation consecutive. Hence, a pair of unordered forward and inverse Fourier Transforms computed using a decimation-in-time and a decimation-in-frequency FFT can use the same twiddle factors, stored in exactly the same way in the distributed memory.

An implementation of the Cooley-Tukey FFT based on multi-sectioning yields perfect arithmetic load balance, while the direct pipelined FFT does not. Hence, even for data allocations where there is no gain in the communication requirements, an FFT based on multi-section has advantages. However, for our implementation on the Connection Machine systems we concluded that the multi-section approach would be inferior. The reason is that the multi-section approach requires data in the processor interchanges to come from different memory locations, which incurs a performance penalty of 30-100% on the Connection Machine systems CM-2 and CM-200, compared to the direct pipelined FFT algorithm. The decrease in communication performance is in most cases greater than, or approximately equal to, the gain from an increased computational efficiency in the kernels evaluating splitting formulas.

Though a radix-2 FFT was chosen for the FFT stages requiring communication, a mix of radix-2, 4 and 8 kernels are used for stages local to each processor. The peak performance of our implementation of the complex-to-complex FFT on the Connection Machine system CM-200 is 12.9 Gflops/s in 32-bit precision, and 10.7 Gflops/s in 64-bit precision for unordered transforms. The corresponding data for ordered transforms is 11.1 Gflops/s and 8.5 Gflops/s, respectively. The peak performance for unordered two-dimensional transforms distributed

over all processors is 5.0 Gflops/s in 32-bit precision and 3.2 Gflops/s in 64-bit precision. The corresponding execution rates for the ordered transforms are 2.7 and 1.5 Gflops/s, respectively. The execution rates for large one-dimensional transforms is slightly higher, and for three-dimensional transforms slightly lower.

References

[1] Jean-Philippe Brunet and S. Lennart Johnsson. All-to-all broadcast with applications on the Connection Machine. *International Journal of Supercomputer Applications*, 6(3):241–256, 1992.

[2] James C. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series. *Math. Comp*, 19:291–301, 1965.

[3] Alan Edelman. Optimal matrix transposition and bit-reversal on hypercubes: All-to-all personalized communication. *Journal of Parallel and Distributed Computing*, 11(4):328–331, 1991.

[4] D. Fraser. Array permutations by index-digit permutation. *J. ACM*, 22:298–308, 1976.

[5] W. Morven Gentleman and G. Sande. Fast Fourier transforms – for fun and profit. In *Proceedings – Fall Joint Computer Conference, 1966*, pages 563–578. AFIPS, 1966.

[6] J.W. Hong and H.T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the 13th ACM Symposium on the Theory of Computation*, pages 326–333. ACM, 1981.

[7] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Computing*, 4(2):133–172, April 1987.

[8] S. Lennart Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Statist. Comput.*, 8(3):354–392, May 1987.

[9] S. Lennart Johnsson and Ching-Tien Ho. Matrix transposition on Boolean n-cube configured ensemble architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

[10] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[11] S. Lennart Johnsson and Ching-Tien Ho. Maximizing channel utilization for all-to-all personalized communication on Boolean cubes. In *The Sixth Distributed Memory Computing Conference*, pages 299–304. IEEE Computer Society Press, 1991.

[12] S. Lennart Johnsson and Ching-Tien Ho. Generalized shuffle permutations on Boolean cubes. *J. Parallel and Distributed Computing*, 16(1):1–14, 1992.

[13] S. Lennart Johnsson and Ching-Tien Ho. Boolean cube emulation of butterfly networks encoded by Gray code. *Journal of Parallel and Distributed Computing*, 20(3):261–279, 1994. Department of Computer Science, Yale University, Technical Report, YALEU/DCS/RR-764, February, 1990.

[14] S. Lennart Johnsson, Ching-Tien Ho, Michel Jacquemin, and Alan Ruttenberg. Computing fast Fourier transforms on Boolean cubes and related networks. In *Advanced Algorithms and Architectures for Signal Processing II*, volume 826, pages 223–231. Society of Photo-Optical Instrumentation Engineers, 1987.

[15] S. Lennart Johnsson, Robert L. Krawitz, Douglas MacDonald, and Roger Frye. A radix-2 FFT on the Connection Machine. In *Supercomputing 89*, pages 809–819. ACM, November 1989.

[16] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, 1982.

[17] Alan V. Oppenheimer and Ronald W. Schafer. *Digital Signal Processing.* Prentice-Hall, Englewood Cliffs. NJ, 1975.

[18] L.R. Rabiner and B. Gold. *Theory and Application of Digital Signal Processing.* Prentice-Hall, Englewood Cliffs. NJ, 1975.

[19] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[20] Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[21] Thinking Machines Corp. *CM Fortran Reference Manual, Version 2.1*, 1993.

[22] Charles Tong and Paul N. Swarztrauber. Ordered Fast Fourier transforms on a masively parallel hypercube multiprocessor. *Journal of Parallel and Distributed Computing*, 12(1):50–59, May 1991.